The Art of Visualizing High Dimensional Data

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 2.0, June 2022

Abstract

I discuss different techniques to produce professional data videos, animated GIFs, and other visualizations in Python, using the `pillow` and `moviepy` libraries. Applications include visualizing prediction intervals regardless of the number of features (also called independent variables), supervised classification applied to an infinite dataset, convergence of machine learning algorithms, and animations featuring objects of various sizes moving at various speeds according to various paths. For instance, I show a video simulation of 300 comets circling the sun, to assess the risk of a collision.

The Python libraries in question allow for low-level image processing at the pixel level. This is particularly useful to build ad-hoc, original visualization algorithms. I also discuss optimization: amount of memory required, performance of compression techniques, numpy versus math library, anti-aliasing to depixelate an image, and so on. Some of the videos use the RGBA palette format. This 4-dimensional color encoding (red, green, blue, alpha) allows you to set the transparency level (also called "opacity") when objects overlap. It is particularly useful in models involving mixtures or overlapping groups in supervised classification. In that context, not only it helps with visualizations, but it actually solves the classification problem on its own.

Contents

1 Introduction

I start with Figures 2 and 3. It is a simulation of comets orbiting the sun, at various velocities, with various orbit orientations and eccentricities. The goal is to assess the risk of collisions. The pictures do a poor job at rendering all the dimensions involved. Thus I created two videos, available here (showing the orbits) and here (featuring comet properties and collisions). The videos add far more than one dimension – the time – to understand the mechanics of the system.

The purpose of this article is to introduce you to enriched visualizations, with a focus on animated gifs and videos built in Python. For instance, the comet video can feature several dimensions that are difficult to show in a static picture: the comet locations at any given time, the relative velocity of each comet, the change in velocity (acceleration), the change in comet size when approaching the sun, the comet interactions (the apparent collisions), any change in the orbit (orientation or eccentricity), or any change in composition (the color assigned to a comet). The static images are good at showing the size, orbit path, and comet composition.

We could make it a bit more general and represent the movements in 3D. Regardless we can easily display 17 dimensions.

The 17 dimensions featured in the comet video

- location in space and time (3 or 4 dimensions)
- comet composition or type, and change in composition (2 dimensions, categorical variables)
- comet size and change in size (2 dimensions, categorical/binned variables here)
- comet velocity and acceleration, including change in acceleration (3 dimensions)
- orbit orientation (rotation angle) and eccentricity, and change in these metrics (4 dimensions)
- period of each orbit, and collisions (2 dimensions)
- the number of comets at any given time (1 dimension)

While it is possible to show all these features in traditional time series plots, the video conveys a more compelling message, providing strong visual insights. Note that in my video, the orbit, size and composition of any given comet, is static. I use colors to represent the velocity and eccentricity: red = fast, green = high eccentricity, purple = fast + high eccentricity, white = standard. Also, the size (big or small) is related to the maximum distance to the sun: small dots correspond to comets staying permanently close to the sun.

The remaining of this article focuses on four applications: prediction intervals in any dimension, supervised classification, convergence of algorithms such as gradient descent when dealing with chaotic functions, and spatial time series (the comets illustration). In some of these cases, using a video helps, and in other cases, it does not. All Python visualizations use the RGB (red / green / blue) color model [Wiki]. It can represent 3 dimensions. One of the videos uses the RGBA model [Wiki], allowing you to add transparency or opacity. This is particularly useful when displaying overlapping clusters: when a red cluster overlaps with a green one, the intersection looks yellow (red + green = yellow). I also use anti-aliasing techniques [Wiki] to make contours look smooth instead of pixelated. Finally, I use a compression technique (FFmpeg, [Wiki]) to reduce the size of the animated gifs.

In a future article, I will add a sound track to the video, related to the behavior of the whole system. The sound (amplitude, frequency, texture) can easily add 3 dimensions. For instance, it can represent the local temperature, density and size of the universe at any given time.

2 Applications

I discuss specifics of the code, such as Python instructions and libraries, in section 3. This section focuses on high level concepts, including the math behind the visualizations. As much as possible, I use notations that are compatible with the names of the variables and arrays in the Python code.

2.1 Spatial time series

Here I discuss the comet visualization introduced in section 1. All orbits are elliptic. The orbits are bivariate continuous time series, or in other words, spatial time series. This will become obvious when looking at the parametric equation of the ellipse. The cartesian equation of an unslanted ellipse centered at the origin is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

with $a, b > 0$. The eccentricity is defined as $\epsilon = \sqrt{|a^2 - b^2|}$. It $\epsilon = 0$, the ellipse is a circle. If ϵ is large, the ellipse is elongated. In all cases, an "horizontal" ellipse has two focus points: $(g'_x, g'_y) = (0, \epsilon)$ and $(g_x, g_y) = (0, -\epsilon)$. Without loss of generality due to rotational symmetry, I only consider horizontal ellipses, and I ignore the second focus point. The parametric equation of the ellipse is

$$\begin{aligned} x_0(t) &= a \cdot \cos(vt + \tau), \\ y_0(t) &= b \cdot \sin(vt + \tau), \end{aligned}$$

where v is the speed of the comet, t represents the time, and τ determines the initial position of the comet when $t = 0$. I then apply a rotation of angle θ. The parameter θ is referred to as the orientation of the orbit. Now the parametric equation of the ellipse becomes

$$\begin{aligned} x(t) &= x_0(t)\cos\theta - y_0(t)\sin\theta, \\ y(t) &= x_0(t)\sin\theta + y_0(t)\cos\theta, \end{aligned}$$

and its focus point of interest becomes $(g_x, g_y) = (g'_x \cos\theta - g'_y \sin\theta, g'_x \sin\theta + g'_y \cos\theta)$.

Now, I want the sun to be at the origin, and have the comet rotates around the sun. This is accomplished by subtracting the vector (g_x, g_y) to $(x(t), y(t))$. Finally, there are m comets labeled $0, \ldots, m-1$. The notation $(x_n(t), y_n(t))$ denotes the position of the n-th comet at time t, with $0 \leq n < m$. Time is sampled evenly: each sample value produces a frame in the video, with m dots: one per comet.

The parameters τ, θ, v, a, b, more precisely $\tau_n, \theta_n, v_n, a_n, b_n$ as there is one set for each orbit, are randomized. However, for a realistic simulation that complies with the laws of the universe (for instance, Kepler's laws), several constraints should be put on these parameters. For instance, the speed should increase when approaching the sun. Also the gravitational interactions are ignored. In fact, the real orbits are not truly periodic because of this, unlike in the simulations.

2.2 Prediction intervals in any dimensions

This application, including the accompanying Python code and Excel spreadsheet, is discussed in detail in my article on fuzzy regression [3], using synthetic data. There is no need for a video here: a simple scatterplot will do. I included this visualization because it works in any dimension, and it is rarely if ever mentioned elsewhere, despite its ease of interpretation. Also, it can be done in Excel: see Figure 1, and the Excel implementation `fuzzy4.xlsx`, available here.

The purpose is to compute predictions and prediction intervals for data outside of a training set, typically in a regression problem (linear or not). I use a validation set [Wiki] to assess performance, comparing the true value with the predicted one. The validation set is a subset of the training set, not used to train the model, but rather, to test it. The observed response Z_{obs} – also called true value – depends on m features $X_1, \ldots, X_m$. All are column vectors, with each entry corresponding to an observation. Thus, the dimension is m. The predicted value at a specific location $x = (x_1, \ldots, x_m)$ in the m-dimensional feature space is denoted as $Z_{\text{pred}}(x)$, while the lower and upper bounds of (say) a 90% prediction interval are denoted as $Z_{.05}(x)$ and $Z_{.95}(x)$ respectively.

If $m > 2$, visualizing the prediction intervals becomes challenging. One way to do it is to create a scatterplot featuring the bivariate vectors $(Z_{\text{obs}}(x), Z_{\text{pred}}(x))$ colored in blue, for all x in the validation set. Thus $Z_{\text{obs}}(x)$ is on the horizontal axis, and $Z_{\text{pred}}(x)$ on the vertical axis. Then, add the points $(Z_{\text{obs}}(x), Z_{.95}(x))$ in red, and the points $(Z_{\text{obs}}(x), Z_{.05}(x))$ in green, on the scatterplot. The end result is Figure 1. Of course, it works regardless of the dimention.

Interpreting the visualization is easy. If the observed and predicted values were identical, the point cloud, that is, the $(Z_{\text{obs}}(x), Z_{\text{pred}}(x))$'s, should all be located on the main diagonal. Deviations on the vertical axis from the main diagonal show the individual residual errors. It provides a much better picture of the goodness fit [Wiki] than any single metric such as R-squared or root-mean-squared deviation RMSE [Wiki]. Note that metrics such as R-squared have several drawbacks, and alternatives are discussed in [3].

Another important feature of the visualization is the slope of the regression line going through the point cloud. If the fit was perfect and all the points aligned on the main diagonal, the slope would be equal to one. In practice, it is always between zero and one. A low slope, say 0.5, does not mean that the fit is bad. It means that the regression is smoothing out the spikes in the data, and acts as noise-removing filter. This is actually a good thing. It is easy to rescale the predicted values so that their variance is identical to that computed on the training set. This will restore the higher variations in the original data, while preserving the smoothness and the R-squared. Indeed, the R-squared, defined as the square of the correlation between the observed and predicted values, is invariant under scaling and/or translations.

2.3 Supervised classification of an infinite dataset

In this problem, the visualization displays 9 dimensions: one for the time (in the video), one for the size of the dots, one for the category or group label, two for the physical location (state space), and four for the RGBA colors. The last frame of the video, showing raw data, is pictured in Figure 5. The horizontal red line is the real axis (the X-axis). The black dot on the left, on the red line, is the origin $(0, 0)$ and the black dot on the right corresponds to $(1, 0)$. A version with bigger dots to actually perform the fuzzy classification of the whole state space is pictured in Figure 6. This is the most insightful visualization in this case. The corresponding videos are found respectively here and here. Notice the huge overlap between the three groups (red, blue, yellow). Yet strong patterns emerge: the points are not randomly distributed.

The dataset comes from number theory. It is not synthetic in the sense that it represents a real phenomenon. Yet, if it was possible to use the whole, infinite dataset, the boundary of the clusters, the boundary of the holes, and the extend of the cluster overlap, would be known exactly. Theoretical considerations allow to solve some of the mysteries. But the problem investigated here is related to the Riemann Hypothesis [Wiki], one of the most famous unsolved mathematical problems of all times. So machine learning techniques are still useful to gain

more insights, and do a great job here. In math circles, the methods used here are described as experimental mathematics [Wiki].

2.3.1 Machine learning perspective

Before diving into the mathematical details, I explain the machine learning aspects of the problem. Color levels in each channel (red, green, blue) are represented by integer values between 0 and 255. The use of the RGBA color model helps visualize cluster overlap. Regions with a high density of yellow points and low density of blue points appear somewhat greenish, but with more yellow than blue in the color. Conversely, regions with a low density of yellow points and high density of blue points also appear somewhat greenish, but with more blue than yellow in the color. High point density results in brighter regions, while low density regions are almost transparent.

Indeed, the supervised classification is automatically performed based on that mechanism alone, with the size of a dot being the main hyperparameter. To find the label (red, yellow or blue) assigned to any location, one has to get its RGB color in Figure 6. For instance, if RGB $= (155, 105, 100)$ then the probability that the point is red, blue or yellow is respectively $50/255, 100/255$ and $105/255$. See Exercise 1 for this computation. Thus the name fuzzy classification sometimes used, but it is actually quite similar to Bayesian classification.

This is achieved thanks to the A component in the RGBA color scheme: it stands for the opacity or transparency of the color, allowing for color blending via the α-compositing algorithm [Wiki]. The letter A in RGBA is named after the α in question, and this component is sometimes referred to as the α channel. The technique also allows you to easily discriminate between areas of high density and low density, determined by the brightness, that is, the cumulated transparency level computed over overlapping dots. In short, it performs density estimation on its own!

For similar applications, see the "glowing plot" in my book on stochastic processes [4], page 25 (figure 10). Also see the fractal GPU-based classification technique described in the same book pages 23-24, and its related video here. Compared to t-SNE [Wiki] (see also here) my technique may require more memory, especially if extended to 3D, but it does not compute any distance. Thus it is computationally more efficient if the number of data points is large: t-SNE (an unsupervised machine learning technique) requires the computation of all inter-point distances. Also, my method is not a substitute to PCA: it does not perform dimension reduction, only visualization and low-dimension classification at this time. For a new approach to dimension reduction, see my article on linear algebra [2].

Finally, the data points are located on a non-periodic orbit that over time covers a dense area. So, the data points in Figure 5 and 6 are sampled from that orbit (actually 3 orbits: a red, blue and yellow one), in such a way as to be relatively equally spaced on the orbit. It is possible to obtain perfect spacing using a method similar to the re-weighting technique described in section 3, in my article on shape classification [1]. Without careful sampling, the points are distributed as in Figure 7: the point density is higher where the curvature of the orbit is more pronounced, or when closer to the related hole. It becomes lower on average as the time increases, that is, as more and more video frames are displayed.

The orbits are displayed in Figure 7. The related video can be found here. The added value provided by the video is that it shows how the points slowly fill a dense area over time. Also note the analogy with the comet video, where the sun plays the role of an attractor, yet comets never cross the sun (at least in the simulation). Here, the hole attached to an orbit plays the role of the sun. But a big difference is that the orbits are non-periodic.

Exercise 1 *Point classification.* A point in Figure 6 has the RGB components $(155, 105, 100)$. What is the chance that it belongs to the red, blue or yellow cluster?

Solution

Let $p_{\text{r}}, p_{\text{b}}, p_{\text{y}}$ be the probabilities in question. They satisfy

$$M^T = \begin{pmatrix} 255 & 0 & 0 \\ 0 & 0 & 255 \\ 255 & 255 & 0 \end{pmatrix}^T \cdot \begin{pmatrix} p_{\text{r}} \\ p_{\text{b}} \\ p_{\text{y}} \end{pmatrix} = \begin{pmatrix} 155 \\ 105 \\ 100 \end{pmatrix}.$$

The solution $p_{\text{r}} = 50/255, p_{\text{b}} = 100/255, p_{\text{y}} = 105/255$ is obtained by solving the above system. The first row in the matrix M corresponds to red, the second one to blue, and the third one to yellow RGB vectors. It there was one more cluster, say turquoise, we would have one extra probability p_{t}. Instead of a system with 3 equations and 3 unknowns, we would have 3 equations with 4 unknowns. Typically it has infinitely many solutions. I will discuss how to proceed in a future article. Of course, if there is no or little cluster overlap, the solution is still

trivial. Also, a pure red point is still assigned to red. A simple solution is to assign the point to the cluster with the closest color. Or you can work with two identical images for pixel locations, but with different RGB colors, to cover up to $3 + 3 = 6$ clusters.

As for the geometrical space (the 2D complex plane in this case), in three dimensions, rather than using an image represented by a matrix of dimension width $\times$ height, one can use a tensor of dimension width $\times$ height $\times$ depth. In this case, each slice of the tensor is a 2D image. In some sense, it is similar to using a video, with each frame representing a 2D image. ∎

2.3.2 Six challenging problems

There are few other interesting machine learning problems worth investigating, raised by Figure 6. I summarize them in the list below.

- Problem 1: The set of yellow points seems to be bounded. Is that also true for the blue and red dots?
- Problem 2: Assuming the set of yellow points is bounded, what is the shape of its boundary?
- Problem 3: There are holes in the blue and yellow point distributions. Can we characterize these holes?
- Problem 4: On the horizontal axis, some segments have no blue dots, some have no yellow dots. Can we characterize these segments?
- Problem 5: If we continue adding points indefinitely, will the holes eventually shrink to empty sets?
- Problem 6: If we continue adding points indefinitely, will the point distributions cover dense areas?

Keep in mind that the set of points is infinite, but only a finite number of points is shown in the picture. Before going into the mathematical details, I will mention this: if you solve Problem 4, you will probably win the Fields Medal in mathematics [Wiki] (the equivalent of the Nobel Prize of mathematics), and a $1 million award for solving one of the seven Millennium Problems [Wiki].

Partial solutions to the six problems are discussed in section 2.3.4.

2.3.3 Mathematical background: the Riemann Hypothesis

What I call data points are values of the Dirichlet eta function $\eta(\sigma + it)$ [Wiki] computed at sampled valued of $t > 0$, for $m = 3$ values of σ, namely $\sigma_0 = 0.50$ corresponding to the red dots, $\sigma_1 = 0.75$ corresponding to the blue dots, and $\sigma_2 = 1.25$ corresponding to the yellow dots. The notation $\sigma + it$ for the complex argument is well established in number theory, and I decided to keep it. There are mathematicians interested in the problem who will read this article, and using a different notation would make my presentation awkward and possibly confusing to them.

The η function returns a complex value defined by

$$\begin{aligned} \Re[\eta(\sigma + it)] &= \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{\cos(t \log k)}{k^{\sigma}}, \\ \Im[\eta(\sigma + it)] &= \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{\sin(t \log k)}{k^{\sigma}}, \end{aligned} \tag{1}$$

where $\Re, \Im$ represent respectively the real and imaginary parts. For our purpose, no knowledge of complex number theory is required. The real and imaginary parts are just the two components of a 2D vector, with the real part on the horizontal axis (X-axis), and the imaginary part on the vertical axis (Y-axis). The notations used in the Python code in sections 3.2 and 3.3, for the real and imaginary part of the η function, are respectively `etax[n]` and `etay[n]`. Here $n = 0$ corresponds to $\sigma = \sigma_0$, $n = 1$ to $\sigma = \sigma_1$ and $n = 2$ to $\sigma = \sigma_2$. The time argument t is a global variable in the Python code, incremented at each iteration, starting with $t = 0$.

Figures that show the orbit are based on fixed increments $\Delta t = 0.04$. Figures showing the points but not the orbit use variable increments. This is to correct for the fact that fixed increments do not produce points evenly spaced on the orbit. In that case, a separate timer is used for each value of σ. In the Python code in section 3.3, it corresponds to `t[0]`, `t[1]`, and `t[2]` respectively for the red, blue and yellow points. The main loop is over the time, and the inner loop is over the three colors (that is, the three values of σ). The η function is computed by the function `G` in the Python code, returning both the real and imaginary parts. It uses the first 10,000 terms of the sums in formula (1). These are slow converging series. I discuss convergence acceleration techniques, chaotic convergence, and numerical accuracy in section 2.4.

2.3.4 Partial solutions to the six challenging problems

In machine learning, typically no mathematical proof is available to show that a model is exact. For instance, statistical models show that smoking increases the chances of getting lung cancer. The arguments are compelling. But there is no formal proof to this. Typically, it is difficult to establish causality. To the contrary, in mathematics, usually formal proofs are available, and they can be used to test and benchmark statistical models. One would think that there is a precise, mathematical answer to the six problems raised in section 2.3.1. Unfortunately, this is not the case here. However, some partial answers are available. First, let me define what I mean by "hole".

Definition. *The T-hole* $\Omega_T = \Omega_T(\sigma)$ *is the largest circle centered at some location* t_T *on the real (horizontal) axis, for which* $\eta(\sigma + it) \notin \Omega_T$ *if* $0 < t \leq T$. *The hole* Ω *is the limit of* Ω_T, *as* $T \to \infty$.

In short, η's orbit, given σ, never enters the hole if $0 < t < T$. Here $t_T = t_T(\sigma)$ is a function of σ, with $0 \leq t_T < 2$. Another important concept (see problem 4) is the largest segment on the real axis, that is never crossed or hit by the orbit.

Now, let's focus on answering problems 1 – 6. First, the series in formula (1) converge absolutely [Wiki] if $\sigma > 1$. Thus the yellow orbit is bounded: this is a trivial fact. The fact that the blue and red orbits are unbounded was proved long ago. See for instance the classic reference "The Theory of the Riemann-Zeta Function" [5]. This provides a full answer to problem 1.

Then, it is also known that the orbits cover dense areas. And as you keep adding more and more points (thus increasing T), the holes eventually shrink to a set of Lebesgue measure zero: this is true if $0.5 < \sigma < 1$. It is a consequence of the universality property of the Riemann zeta function [Wiki]. This provides a partial answer to problems 5 and 6. If σ is fixed and t is bounded, I presume that the hole associated to the blue and yellow orbits always exist. For the blue orbit, this statement, especially if applied to all σ in $]0.5, 1[$, is stronger than the Riemann Hypothesis (RH), and thus unproven to this day. Some argue that RH may be unprovable, but that's another story.

The red orbit ($\sigma = 0.5$) has no hole. As σ is decreased from 1.25 to 0.5, the hole shrinks and move to the left on the real axis, towards the origin. It eventually shrinks to an empty set and becomes and attractor point when $\sigma = 0.5$. This is confirmed by the fact that the Riemann zeta function has infinitely many non-trivial zeros if $\sigma = 0.5$. And the roots of the Dirichlet eta and Riemann zeta functions are identical when $0.5 < \sigma < 1$. The hole of the blue orbit seems to encompass the origin, suggesting that if $\sigma = 0.75$, the Riemann zeta function has no zero. To this day, this conjecture, much weaker than RH, is unproven.

The concepts and repulsion or attraction basin [Wiki] is fundamental in dynamical systems. A hole is a repulsion basin, while the origin, for the red orbit, is an attractor point. One of my upcoming books will discuss these topics in detail, for a wide range of dynamical systems. Another nice video featuring the progress of the orbit can be found here. The Python code `image3R_orbit_enhanced.py` to generate it, can be found here.

2.4 Algorithms with chaotic convergence

We are all familiar with pictures, even animated gifs, showing the gradient descent or some other optimization algorithm in action, converging to an optimum depending on initial conditions. In all cases, the surface (be it 2D or 3D) is smooth, though there are numerous examples with several local maxima and minima. See example, with Mathematica code, here. The Wikipedia entry for the gradient descent (here) also features a nice 3D video.

Here I visually illustrate the convergence mechanism when the function much less smooth, in a general context. It is not optimization-related, unless you consider accelerating the convergence to be an optimization problem. Each new frame in the video shows the progress towards the final solution, given initial conditions. The convergence path, for this 2D problem, for six different initial conditions, is shown in Figure 8. It's hard to tell where it starts and where it ends. This is straightforward if you look at the video, here.

The algorithm pictured in Figure 8 computes the successive values of the η function in the complex plane, using formula (1). Each iteration adds one new term to the summation, and generates a new frame. It is possible to significantly improve the speed of convergence, using convergence acceleration techniques [Wiki] such as Euler's transform [Wiki], described page 65 in my book [4]. I explain in my book (same page) why the convergence is so chaotic in this case. The same convergence acceleration techniques apply to gradient descent and other similar iterative algorithms, when successive iterations generate oscillating values. For the η function, Borwein's method [Wiki] may be the best approach.

I tested the numerical stability of the computations by introducing stochastic noise in formula (1). I describe the methodology in the section "Perturbed version of the Riemann Hypothesis", page 20 in my book [4]. The holes described in section 2.3.4 in this paper are very sensitive to minuscule errors in the computations, and are non-existent when very small changes are introduced. This confirms that there is something really unique to the Riemann zeta function. For solutions to optimize highly chaotic functions (compute a global maximum

or minimum), see my book [4] pages 17–18. I used a diverging fixed-point algorithm [Wiki] that emits a signal when approaching a global optimum. The technique will be described in detail in an upcoming article. A quick overview of the methodology is available here.

Finally, the visualization (Figure 8 or the corresponding video here) uses a very large number of RGB colors. To produce the visual effect, I used sine functions to generate the colors. See section 3.2, and page 85 in my book [4] for more details. Palette optimization (see here) will be the subject of an upcoming article.

3 Python code

The Python code, videos, and animated gifs are available on my GitHub repository, here. The videos are also on YouTube, here. For convenience, the Python code is also included in this article. Top variables include `ShowOrbit` (set to true if you want to display the orbit, not just the points), `dot` (the size of the dots), `r` (when iterating over time, it outputs a video frame once every r iterations), `width` and `height` (the dimension of the image). The final image is eventually reduced by half due to the anti-aliasing procedure used to depixelate the curves. This is performed within `img.resize` in the code, using the `Image.LANCZOS` parameter [Wiki].

Ellipses and lines are produced using the Pillow library and its `ImageDraw` functions. Animated gifs are produced either with `images[0].save` (resulting in massive files), or with `videoClip.write_gif`, using the Moviepy library with the ffmpeg parameter [Wiki] for compression. When working with a large number of colors, ffmpeg causes considerable quality loss, not in the rendering of the shapes, but in the color palette. I suggest to use libraries other than Pillow to produce animated gifs, for instance openCV. Eventually, I converted some of the MP4 videos to gifs using the online tool ezgif, also based on ffmpeg.

Reducing the size of the image and the number of frames per second (FPS) will optimize the code and produce much smaller gifs. The biggest improvement, in terms of speed, is replacing all numpy calls (`np.log`, `np.cos` and so on) by math calls (`math.log`, `math.cos` and so on). If you use numpy for image production rather than Pillow, the opposite may be true (I did not test). Finally, the opacity level in the RGBA color model should be set to 127. Currently, it is set to 80; see the fourth parameter for instance in `colp.append((0,0,255,80))`.

3.1 Paths simulation

On GitHub: `image2.py`. Produces the comet video. Description in section 2.1.

```
from PIL import Image, ImageDraw # ImageDraw to draw ellipses etc.
import moviepy.video.io.ImageSequenceClip # to produce mp4 video
from moviepy.editor import VideoFileClip  # to convert mp4 to gif
import numpy as np
import math
import random
random.seed(100)

#--- Global variables ---

m=300          # number of comets
nframe=1500    # number of frames in video
ShowOrbit=True # do not show orbit (default)

count=0        # frame counter

count1=0
count2=0
count3=0
count4=0

width = 1600
height = 1200

a=[]
b=[]
gx=[]    # focus of ellipse (x coord.)
gy=[]    # focus of ellipse (y coord.)
theta=[] # rotation angle of ellpise (the orbit)
v=[]     # speed of comet
```

```
tau=[]   # position of comet on the orbit path, at t = 0
col=[]   # RGB color of the comet
size=[]  # size of the comet
flist=[] # filenames of the images representing each video frame

a=list(map(float,a))
b=list(map(float,b))
gx=list(map(float,gx))
gy=list(map(float,gy))
theta=list(map(float,theta))
v=list(map(float,v))
tau=list(map(float,tau))
flist=list(map(str,flist))

#--- Initializing comet parameters ---

for n in range (m):
  a.append(width*(0.1+0.3*random.random()))
  b.append((0.5+1.5*random.random())*a[n])
  theta.append(2*math.pi*random.random())
  tau.append(2*math.pi*random.random())
  if a[n]>b[n]:
    gyy=0.0
    gxx=math.sqrt(a[n]*a[n]-b[n]*b[n]) # should use -gxx 50% of the time
  else:
    gyy=math.sqrt(b[n]*b[n]-a[n]*a[n]) # should use -gyy 50% of the time
    gxx=0.0
  gx.append(gxx*np.cos(theta[n])-gyy*np.sin(theta[n]))
  gy.append(gxx*np.sin(theta[n])+gyy*np.cos(theta[n]))
  if random.random() < 0.5:
    v.append(0.04*random.random())
  else:
    v.append(-0.04*random.random())
  if abs(a[n]*a[n]-b[n]*b[n])> 0.15*width*width:
    if abs(v[n]) > 0.03:
    # fast comet with high eccentricity
      red=255
      green=0
      blue=255
      count1=count1+1
    else:
    # slow comet with high eccentricity
      red=0
      green=255
      blue=0
      count2=count2+1
  else:
    if abs(v[n]) > 0.03:
    # fast comet with low eccebtricity
      red=255
      green=0
      blue=0
      count3=count3+1
    else:
    # slow comet with low eccentricity
      red=255
      green=255
      blue=255
      count4=count4+1
  col.append((red,green,blue))
  if ShowOrbit:
     size.append(1)
  else:
    if min(a[n],b[n]) > 0.3*width: # orbit with large radius
      size.append(8)
    else:
```

```
    size.append(4)

sunx=int(width/2)  # position of the sun (x)
suny=int(height/2) # position of the sun (y)
if ShowOrbit:
  img = Image.new( mode = "RGB", size = (width, height), color = (0, 0, 0) )
  pix = img.load()
  draw = ImageDraw.Draw(img)
  draw.ellipse((sunx-16, suny-16, sunx+16, suny+16), fill=(255,180,0))

#--- Main Loop ---

for t in range (0,nframe,1): # loop over time, each t corresponds to a ideo frame
  print("Building frame:",t)
  if not ShowOrbit:
    img = Image.new( mode = "RGB", size = (width, height), color = (0, 0, 0) )
    pix = img.load()
    draw = ImageDraw.Draw(img)
    draw.ellipse((sunx-16, suny-16, sunx+16, suny+16), fill=(255,180,0))
  for n in range (m): # loop over asteroid
    x0=a[n]*np.cos(v[n]*t+tau[n])
    y0=b[n]*np.sin(v[n]*t+tau[n])
    x=x0*np.cos(theta[n])-y0*np.sin(theta[n])
    y=x0*np.sin(theta[n])+y0*np.cos(theta[n])
    x=int(x+width/2 -gx[n])
    y=int(y+height/2-gy[n])
    if x >= 0 and x < width and y >=0 and y < height:
      draw.ellipse((x-size[n], y-size[n], x+size[n], y+size[n]), fill=col[n])
  count=count+1
  fname='imgpy'+str(count)+'.png'

  # anti-aliasing mechanism
  img2 = img.resize((width // 2, height // 2), Image.LANCZOS) # anti-aliasing
  # output curent frame to a png file
  img2.save(fname,optimize=True,quality=30)
  flist.append(fname)

clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=20)
# output video file
clip.write_videofile('videopy.mp4')
# output gif image [converting mp4 to gif with ffmpeg compression]
videoClip = VideoFileClip("videopy.mp4")
videoClip.write_gif("videopy.gif",program='ffmpeg') #,fps=2)

print("count 1-4:",count1,count2,count3,count4)
```

3.2 Visual convergence analysis in 2D

On GitHub: `image2R.py`. Produces the successive approximations to the η function. Description in section 2.4.

```
from PIL import Image, ImageDraw # ImageDraw to draw ellipses etc.
import moviepy.video.io.ImageSequenceClip # to produce mp4 video
from moviepy.editor import VideoFileClip  # to convert mp4 to gif
import numpy as np
import math
import random
random.seed(100)

#--- Global variables ---

m=6           # number of curves
nframe=4000   # number of images
count=0       # frame counter
start=2000    # must be smaller than nframe
r=20          # one out of every r image is included in the video
```

```
width = 3200
height =2400

etax=[]
etay=[]
sigma=[]
t=[]
x0=[]
y0=[]
flist=[] # filenames of the images representing each video frame

etax=list(map(float,etax))
etay=list(map(float,etay))
sigma=list(map(float,sigma))
t=list(map(float,t))
x0=list(map(float,x0))
y0=list(map(float,y0))
flist=list(map(str,flist))

#--- Initializing comet parameters ---

for n in range (0,m):
  etax.append(1.0)
  etay.append(0.0)
  t.append(5555555+n/10)
  sigma.append(0.75)
sign=1

minx= 9999.0
miny= 9999.0
maxx=-9999.0
maxy=-9999.0

for n in range (0,m):
  sign=1
  sumx=1.0
  sumy=0.0
  for k in range (2,nframe,1):
    sign=-sign
    sumx=sumx+sign*np.cos(t[n]*np.log(k))/pow(k,sigma[n])
    sumy=sumy+sign*np.sin(t[n]*np.log(k))/pow(k,sigma[n])
    if k >= start:
      if sumx < minx:
        minx=sumx
      if sumy < miny:
        miny=sumy
      if sumx > maxx:
        maxx=sumx
      if sumy > maxy:
        maxy=sumy
sign=1
rangex=maxx-minx
rangey=maxy-miny

img = Image.new( mode = "RGB", size = (width, height), color = (255, 255, 255) )
pix = img.load()
draw = ImageDraw.Draw(img)

red=255
green=255
blue=255
col=(red,green,blue)
count=0

#--- Main Loop ---
```

```
for k in range (2,nframe,1): # loop over time, each t corresponds to a ideo frame
  if k%10 == 0:
    print("Building frame:",k)
  sign=-sign
  for n in range (0,m): # loop over curves
    x0.insert(n,int(width*(etax[n]-minx)/rangex))
    y0.insert(n,int(height*(etay[n]-miny)/rangey))
    etax[n]=etax[n]+sign*np.cos(t[n]*np.log(k))/pow(k,sigma[n])
    etay[n]=etay[n]+sign*np.sin(t[n]*np.log(k))/pow(k,sigma[n])
    x=int(width*(etax[n]-minx)/rangex)
    y=int(height*(etay[n]-miny)/rangey)
    shape = [(x0[n], y0[n]), (x, y)]
    red = int(255*0.9*abs(np.sin((n+1)*0.00100*k)))
    green= int(255*0.6*abs(np.sin((n+2)*0.00075*k)))
    blue = int(255*abs(np.sin((n+3)*0.00150*k)))

    if k>=start:
      # draw line from (x0[n],y0[n]) to (x_new,y_new)
      draw.line(shape, fill =(red,green,blue), width = 1)

  if k>=start and k%r==0:
    fname='imgpy'+str(count)+'.png'
    count=count+1
    # anti-aliasing mechanism
    img2 = img.resize((width // 2, height // 2), Image.LANCZOS) # anti-aliasing
    # output curent frame to a png file
    img2.save(fname) # write png image on disk
    flist.append(fname) # add its filename (fname) to flist

# output video file
clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=20)
clip.write_videofile('riemann.mp4')
```

3.3 Supervised classification

GitHub version: image3R_orbit.py. Produces the orbits of the η function. Description in section 2.3.

```
from PIL import Image, ImageDraw # ImageDraw to draw ellipses etc.
import moviepy.video.io.ImageSequenceClip # to produce mp4 video
from moviepy.editor import VideoFileClip  # to convert mp4 to gif
import numpy as np
import math
import random
random.seed(100)

#--- Global variables ---

m=3           # number of orbits (one for each value of sigma)
nframe=20000  # number of images created in memory
ShowOrbit=True
ShowDots=False
count=0       # frame counter
r=50          # one out of every r image is included in the video
dot=2         # size of a point in the picture
step=0.04     # time increment in orbit

width = 3200  # width of the image
height =2400  # length of the image

#images=[] # to produce the Gif image
etax=[]  # real part of Dirichlet eta function
etay=[]  # imaginary part of Dirichlet eta function
sigma=[]# real part of argument of Dirichlet eta
x0=[]    # previous value of etax on last video frame
```

```
y0=[]    # previous value of etay on last video frame
colp=[] # RGBA color of the point or orbit
t=[]     # time (that is, time in orbit)
flist=[]# filenames of the images representing each video frame

etax=list(map(float,etax))
etay=list(map(float,etay))
sigma=list(map(float,sigma))
x0=list(map(float,x0))
y0=list(map(float,y0))
t=list(map(float,t))
flist=list(map(str,flist))

#--- Eta function ---

def G(tau,sig,nterms):
  sign=1
  fetax=0
  fetay=0
  for j in range(1,nterms):
    fetax=fetax+sign*math.cos(tau*math.log(j))/pow(j,sig)
    fetay=fetay+sign*math.sin(tau*math.log(j))/pow(j,sig)
    sign=-sign
  return [fetax,fetay]

#--- Initializing comet parameters ---

for n in range (0,m):
  etax.append(1.0)
  etay.append(0.0)
  x0.append(1.0)
  y0.append(0.0)
  t.append(0.0)  # start with t=0.0
sigma.append(0.50)
sigma.append(0.75)
sigma.append(1.25)
colp.append((255,0,0,80))
colp.append((0,0,255,80))
colp.append((255,180,0,80))

if ShowOrbit:
  minx=-2
  maxx=3
else:
  minx=-1
  maxx=2
rangex=maxx-minx
rangey=0.75*rangex
miny=-rangey/2
maxy=rangey/2
rangey=maxy-miny

img = Image.new( mode = "RGB", size = (width, height), color = (255, 255, 255) )
# pix = img.load() # pix[x,y]=col[n] to modify the RGB color of a pixel
draw = ImageDraw.Draw(img,"RGBA")

gx=width*(0.0-minx)/rangex
gy=height*(0.0-miny)/rangey
hx=width*(1.0-minx)/rangex
hy=height*(0.0-miny)/rangey
draw.ellipse((gx-8, gy-8, gx+8, gy+8), fill=(0,0,0,255))
draw.ellipse((hx-8, hy-8, hx+8, hy+8), fill=(0,0,0,255))
draw.rectangle((0,0,width-1,height-1), outline ="black",width=1)
draw.line((0,gy,width-1,hy), fill ="red", width = 1)

count=0
```

```
#--- Main Loop ---

for k in range (2,nframe,1): # loop over time, each t corresponds to an image
  if k %10 == 0:
    string="Building frame:" + str(k) + "> "
    for n in range (0,m):
      string=string+ " | " + str(t[n])
    print(string)
  for n in range (0,m): # loop over the m orbits
    if ShowOrbit:
      # save old value of etax[n], etay[n]
      x0.insert(n,width*(etax[n]-minx)/rangex)
      y0.insert(n,height*(etay[n]-miny)/rangey)
    (etax[n],etay[n])=G(t[n],sigma[n],10000) # 500 -> tau
    x= width*(etax[n]-minx)/rangex
    y=height*(etay[n]-miny)/rangey
    if ShowOrbit:
      if k>2:
        # draw line from (x0[n],y0[n]) to (x,y)
        draw.line((int(x0[n]),int(y0[n]),int(x),int(y)), fill =colp[n], width = 0)
        if ShowDots:
          draw.ellipse((x-dot, y-dot, x+dot, y+dot), fill =colp[n])
      t[n]=t[n]+step
    else:
      draw.ellipse((x-dot, y-dot, x+dot, y+dot), fill =colp[n])
      t[n]=t[n]+200*math.exp(3*sigma[n])/(1+t[n]) # 0.02
  if k%r==0: # this image gets included as a frame in the video
    draw.ellipse((gx-8, gy-8, gx+8, gy+8), fill=(0,0,0,255))
    draw.ellipse((hx-8, hy-8, hx+8, hy+8), fill=(0,0,0,255))
    fname='imgpy'+str(count)+'.png'
    count=count+1
    # anti-aliasing mechanism
    img2 = img.resize((width // 2, height // 2), Image.LANCZOS) # anti-aliasing
    # output curent frame to a png file
    img2.save(fname) # write png image on disk
    flist.append(fname) # add its filename (fname) to flist
#   images.append(img2) # to produce Gif image

# output video file
clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=20)
clip.write_videofile('riemann.mp4')

# output gif file - commented out because it is way too large
# images[0].save('riemann.gif',save_all=True, append_images=images[1:],loop=0)
```

4 Visualizations

The videos and animated gifs are available on my GitHub repository, here. The videos are also on YouTube, here. The remaining of this article consists of selected frames from these videos: those that are referenced in the text (figures 1–8).

References

[1] Vincent Granville. Computer vision: Shape classification via explainable AI. *Preprint*, pages 1–7, 2022. MLTechniques.com [Link]. 4

[2] Vincent Granville. Gentle introduction to linear algebra, with spectacular applications. *Preprint*, pages 1–9, 2022. MLTechniques.com [Link]. 4

[3] Vincent Granville. Interpretable machine learning: Multipurpose, model-free, math-free fuzzy regression. *Preprint*, pages 1–11, 2022. MLTechniques.com [Link]. 3

[4] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective*. MLTechniques.com, 2022. [Link]. 4, 6, 7

[5] E.C. Titchmarsh and D.R. Heath-Brown. *The Theory of the Riemann Zeta-Function.* Oxford Science Publications, second edition, 1987. 6

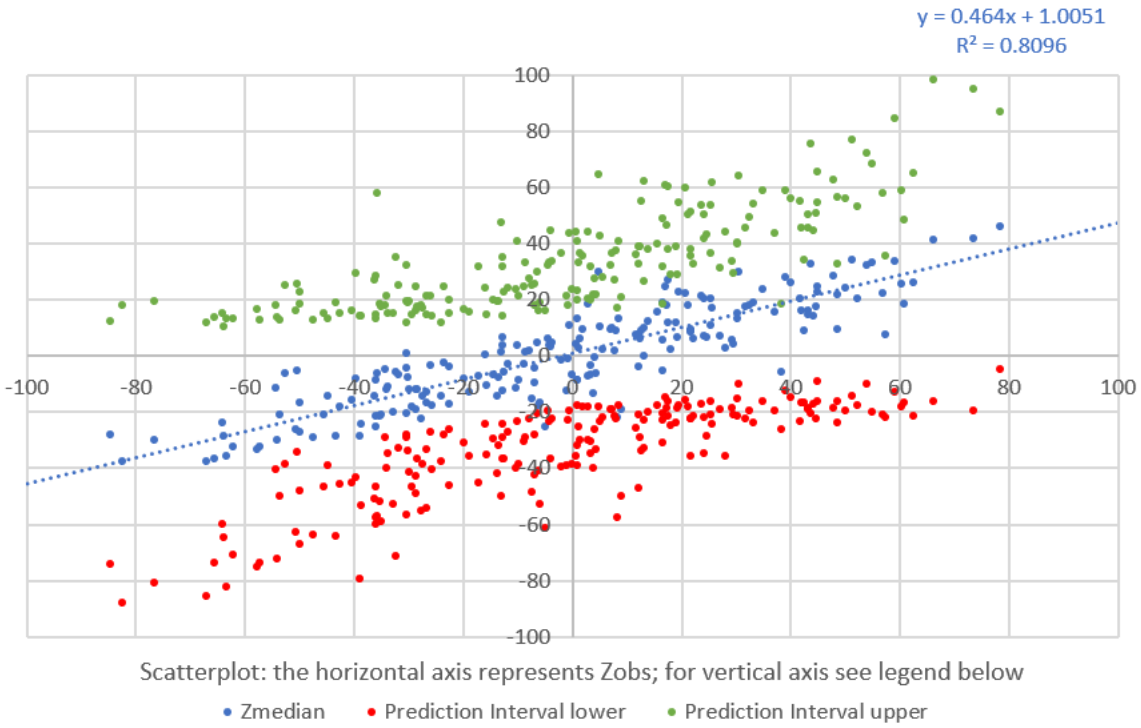

Figure 1: Scatterplot observations vs. predicted values, with prediction intervals (in any dimension)

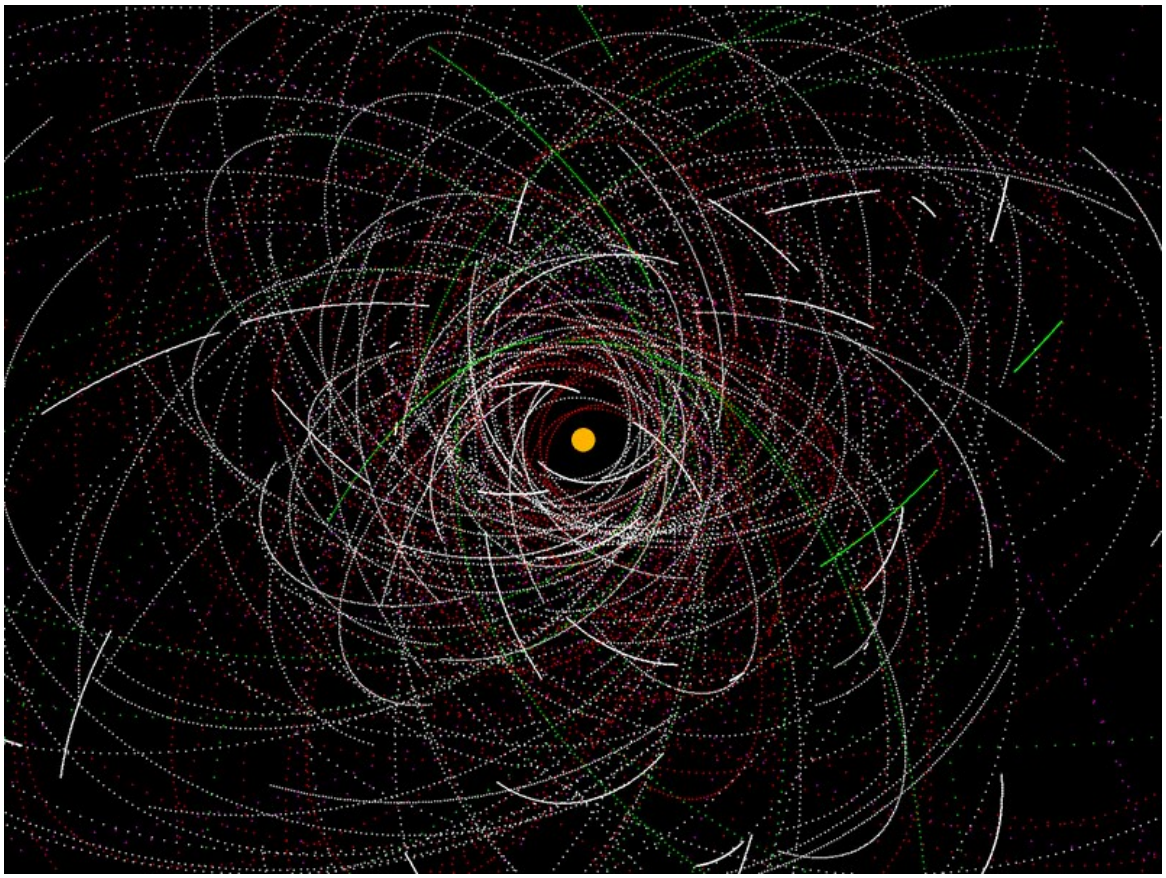

Figure 2: Comets orbiting the sun: simulation

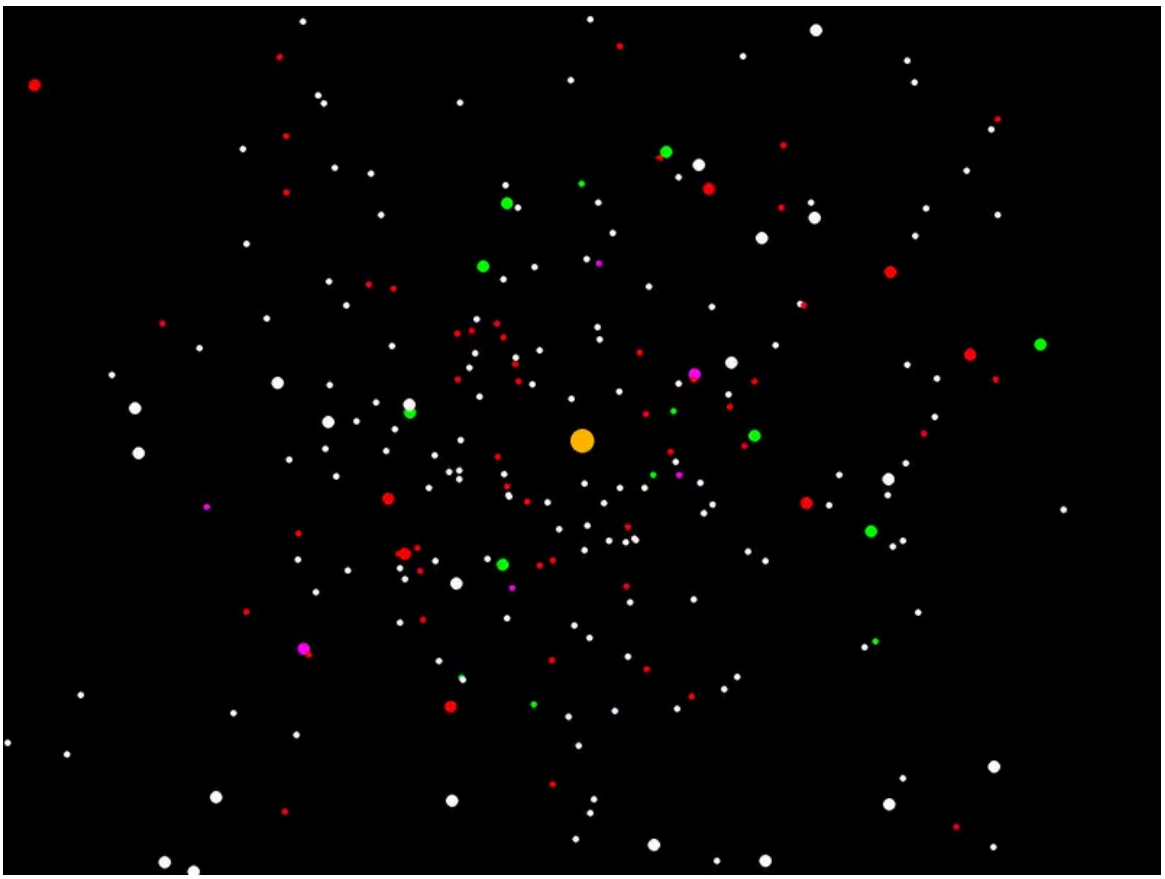

Figure 3: Comets orbiting the sun: snapshot in time

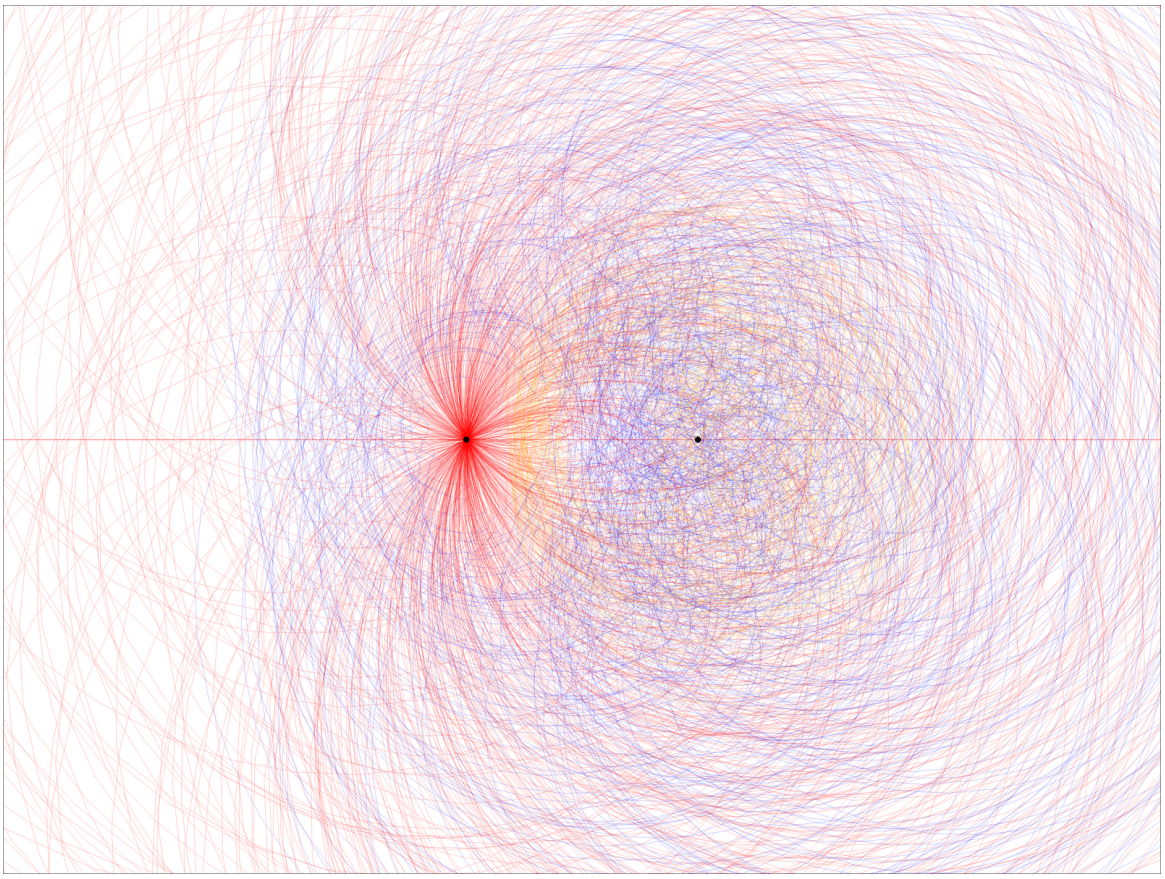

Figure 4: Three orbits of $\eta(\sigma + it)$: $\sigma = 0.5$ (red), 0.75 (blue) and 1.25 (yellow)

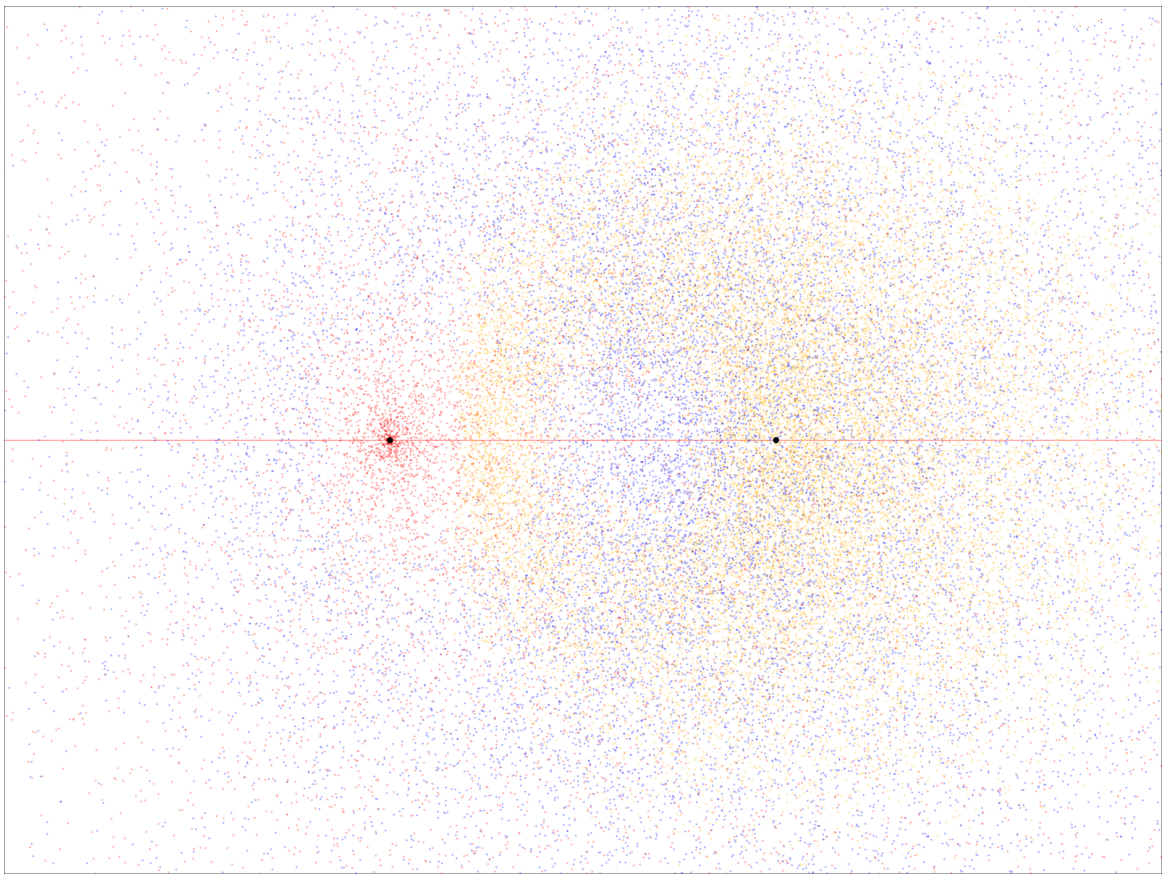

Figure 5: Sample orbit points of $\eta(\sigma + it)$: $\sigma = 0.5$ (red), 0.75 (blue) and 1.25 (yellow)

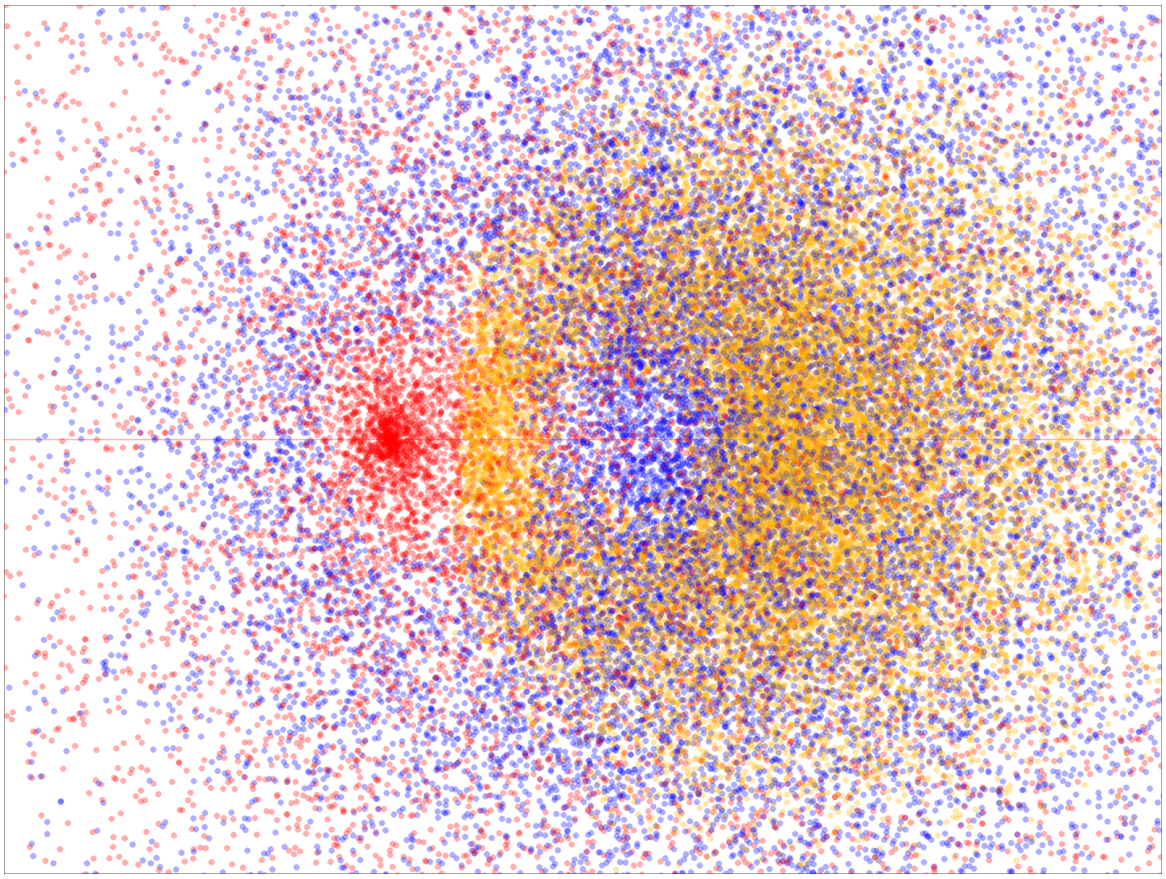

Figure 6: Sample orbit points of $\eta(\sigma + it)$: $\sigma = 0.5$ (red), 0.75 (blue) and 1.25 (yellow)

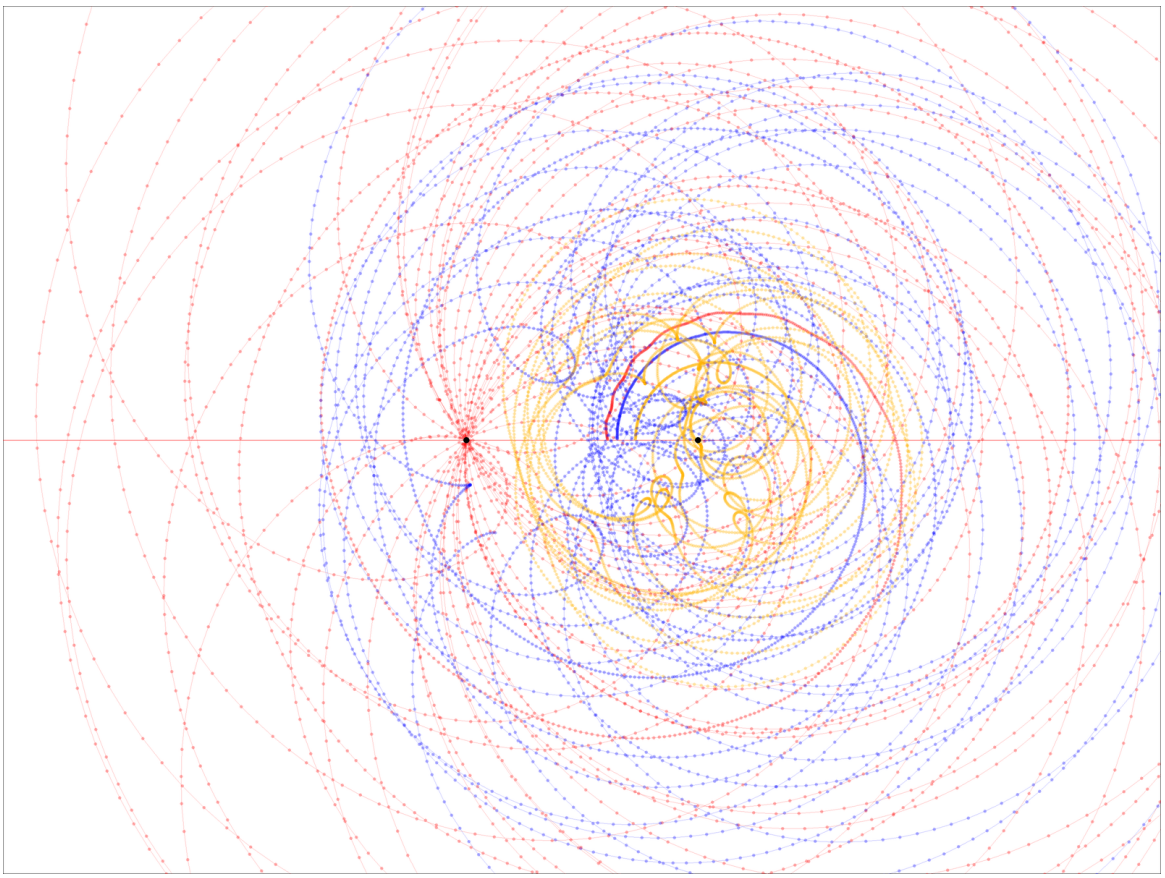

Figure 7: Raw orbit points of $\eta(\sigma + it)$: $\sigma = 0.5$ (red), 0.75 (blue) and 1.25 (yellow)

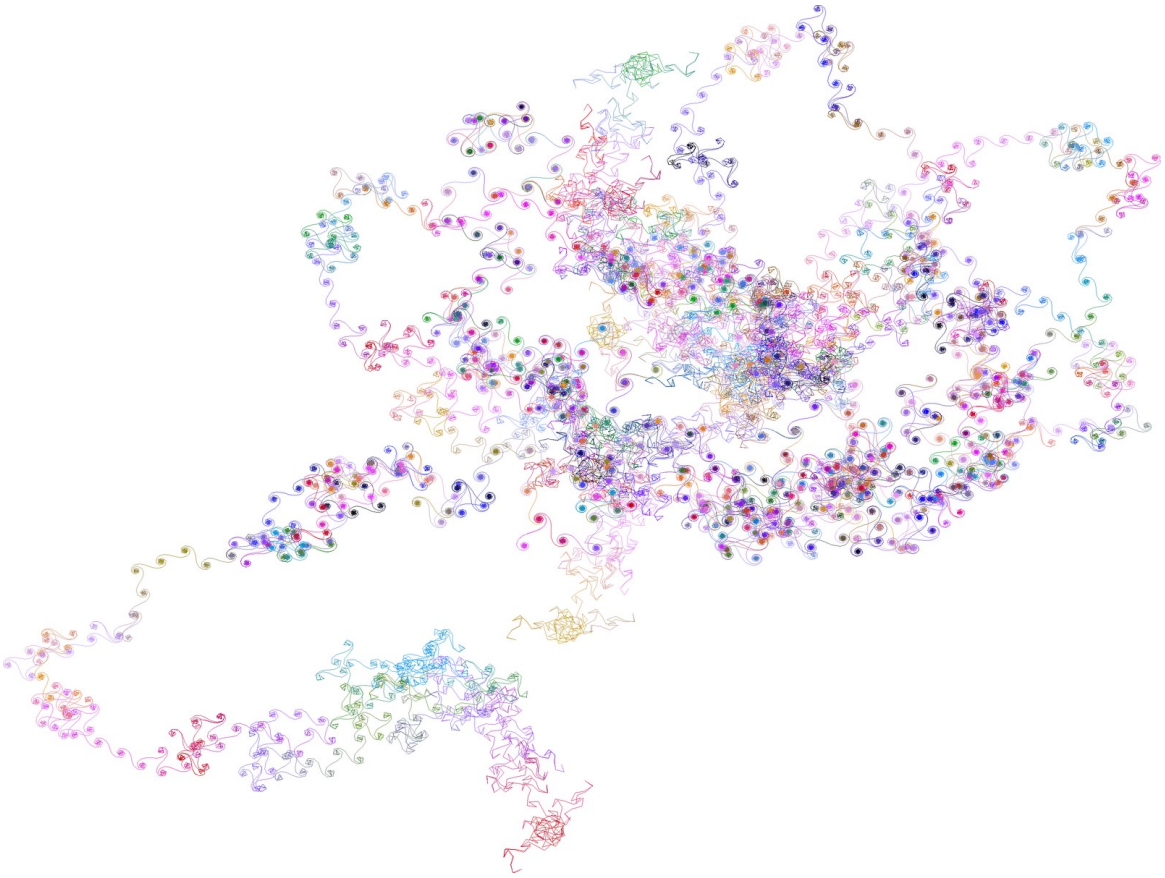

Figure 8: Convergence of partial sums of $\eta(z)$, for six $z = \sigma + it$ in the complex plane